6. Syntax

Learning objectives:

- syntax and semantics
- syntax diagrams and EBNF describe context-free grammars
- terminal and nonterminal symbols
- productions
- definition of EBNF by itself
- parse tree
- grammars must avoid ambiguities
- infix, prefix, and postfix notation for arithmetic expressions
- prefix and postfix notation do not need parentheses

Syntax and semantics

Computer science has borrowed some important concepts from the study of natural languages (e.g. the notions of syntax and semantics). *Syntax* rules prescribe how the sentences of a language are formed, independently of their meaning. *Semantics* deals with their meaning. The two sentences "The child draws the horse" and "The horse draws the child" are both syntactically correct according to the accepted rules of grammar. The first sentence clearly makes sense, whereas the second sentence is baffling: perhaps senseless (if "draw" means "drawing a picture"), perhaps meaningful (if "draw" means "pull"). Semantic aspects—whether a sentence is meaningful or not, and if so, what it means—are much more difficult to formalize and decide than syntactic issues.

However, the analogy between natural languages and programming languages does not go very far. The choice of English words and phrases such as "begin", "end", "goto", "if-then-else" lends a programming language a superficial similarity to natural language, but no more. The possibility of verbal encoding of mathematical formulas into pseudo-English has deliberately been built into COBOL; for example, "compute velocity times time giving distance" is nothing but *syntactic sugar* for "distance := velocity · time". Much more important is the distinction that natural languages are not rigorously defined (neither the vocabulary, nor the syntax, and certainly not the semantics), whereas programming languages should be defined according to a rigorous formalism. Programming languages are much closer to the *formal* notations of mathematics than to natural languages, and *programming notation* would be a more accurate term.

The *lexical part* of a modern programming language [the alphabet, the set of reserved words, the construction rules for the identifiers (i.e. the equivalent to the vocabulary of a natural language) and the *syntax* are usually defined formally. However, system-dependent differences are not always described precisely. The compiler often determines in detail the syntactic correctness of a program with respect to a certain system (computer and operating system). The semantics of a programming language could also be defined formally, but this is rarely done, because formal semantic definitions are extensive and difficult to read.

The syntax of a programming language is not as important as the semantics, but good understanding of the syntax often helps in understanding the language. With some practice one can often guess the semantics from the syntax, since the syntax of a well-designed programming language is the frame that supports the semantics.

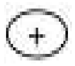
Grammars and their representation: syntax diagrams and EBNF

The syntax of modern programming languages is defined by *grammars*. These are mostly of a type called *context-free grammars*, or close variants thereof, and can be given in different notations. *Backus-Naur form (BNF)*, a milestone in the development of programming languages, was introduced in 1960 to define the syntax of Algol. It is the basis for other notations used today, such as *EBNF (extended BNF)* and graphical representations such as *syntax diagrams*. EBNF and syntax diagrams are syntactic notations that describe exactly the *context-free* grammars of formal language theory.

Recursion is a central theme of all these notations: the syntactic correctness and structure of a large program text are reduced to the syntactic correctness and structure of its textual components. Other common notions include: *terminal* symbol, *nonterminal* symbol, and *productions* or *rewriting rules* that describe how nonterminal symbols generate strings of symbols.

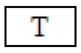
The set of terminal symbols forms the *alphabet* of a language, the symbols from which the sentences are built. In EBNF a terminal symbol is enclosed in single quotation marks; in syntax diagrams a terminal symbol is represented by writing it in an oval:

'+'



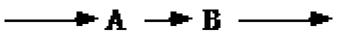
Nonterminal symbols represent syntactic entities: statements, declarations, or expressions. Each nonterminal symbol is given a name consisting of a sequence of letters and digits, where the first character must be a letter. In syntax diagrams a nonterminal symbol is represented by writing its name in a rectangular box:

T



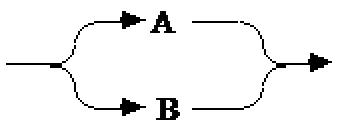
If a construct consists of the catenation of constructs A and B, this is expressed by

AB



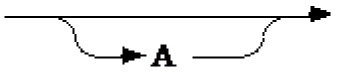
If a construct consists of either A or B, this is denoted by

A|B



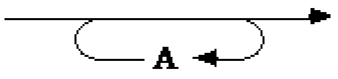
If a construct may be either construct A or nothing, this is expressed by

[A]



If a construct consists of the catenation of any number of A's (including none), this is denoted by

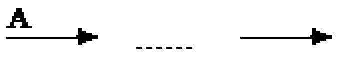
{A}



In EBNF parentheses may be used to group entities [e.g. (A | B)].

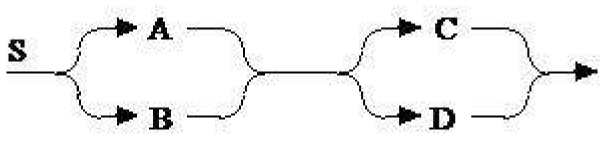
For each nonterminal symbol there must be at least one production that describes how this syntactic entity is formed from other terminal or nonterminal symbols using the composition constructs above:

A = ------ .



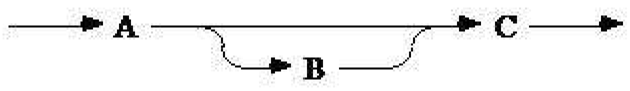
The following examples show productions and the constructs they generate. A, B, C, D may denote terminal or nonterminal symbols.

S = (A | B) (C | D) .



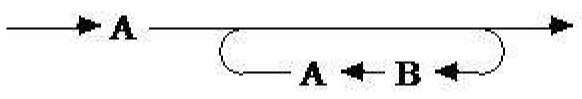
generates: AC AD BC BD

T = A [B] C .

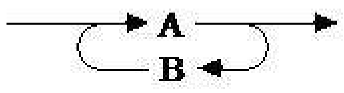


generates: AC ABC

U = A {B A} .



or



generates: A ABA ABABA ABABABA ...

EBNF is a formal language over a finite alphabet of symbols introduced above, built according to the rules explained above. Thus it is no great surprise that EBNF can be used to define itself. We use the following names for syntactic entities:

stmt	A syntactic equation.
expr	A list of alternative terms.
term	A concatenation of factors.
factor	A single syntactic entity or parenthesized expression.
nts	Nonterminal symbol that denotes a syntactic entity. It consists of a sequence of letters and digits where the first character must be a letter.
ts	Terminal symbol that belongs to the defined language's vocabulary. Since the vocabulary depends on the language to be defined there is no production for ts.

EBNF is now defined by the following productions:

EBNF = { stmt } .

stmt = nts '=' expr '.' .

expr = term { '|' term } .

term = factor { factor } .

factor = nts | ts | '(' expr ')' | '[' expr ']' | '{' expr '}' .

nts= letter { letter | digit } .

Example: syntax of simple expressions

The following productions for the three nonterminals E(xpression), T(erm), and F(actor) can be traced back to Algol 60. They form the core of all grammars for arithmetic expressions. We have simplified this grammar to define a class of expressions that lacks, for example, a unary minus operator and many other convenient notations. These details are but not important for our purpose: namely, understanding how this grammar assigns the correct structure to each expression. We have further simplified the grammar so that constants and variables are replaced by the single terminal symbol # (Exhibit 6.1):

```
E  =  T  {  (  '+'  |  '-'  )  T  }  .
T  =  F  {  (  '·'  |  '/'  )  F  }  .
F  =  '#'  |  '('  E  ')'  .
```

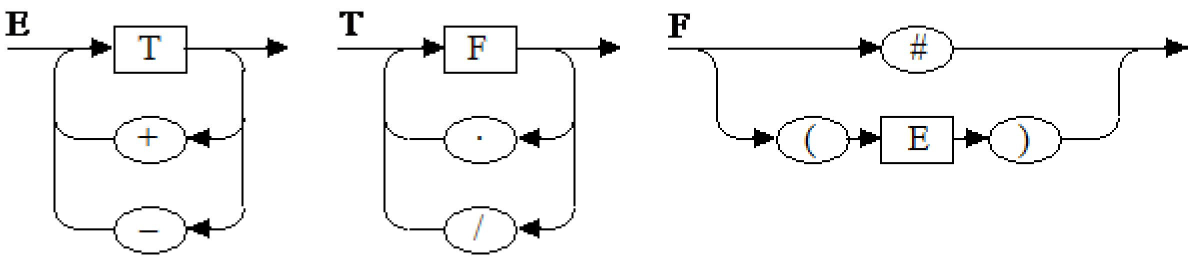


Exhibit 6.1: Syntax diagrams for simple arithmetic expressions.

From the nonterminal E we can *derive* different expressions. In the opposite direction we start with a sequence of terminal symbols and check by *syntactic analysis*, or *parsing*, whether a given sequence is a valid expression. If this is the case the grammar assigns to this expression a unique tree structure, the *parse tree* (Exhibit 6.2).

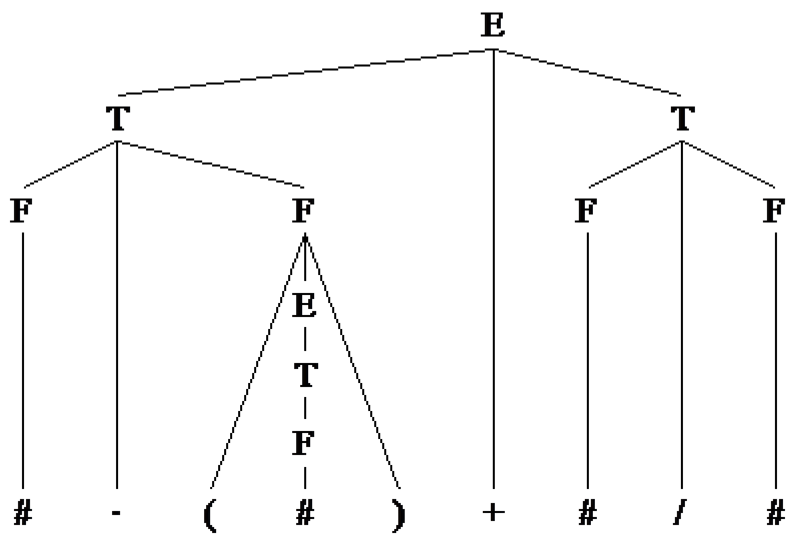


Exhibit 6.2: Parse tree for the expression # · (#) + # / # .

Exercise: syntax diagrams for palindromes

A palindrome is a string that reads the same when read forward or backward. *Examples:* 0110 and 01010. 01 is not a palindrome, as it differs from its reverse 10.

1. What is the shortest palindrome?
2. Specify the syntax of palindromes over the alphabet {0, 1} in EBNF-notation, and by drawing syntax diagrams.

Solution

1. The shortest palindrome is the null or empty string.
2. S = ['0' | '1'] | '0' S '0' | '1' S '1' (Exhibit 6.3).

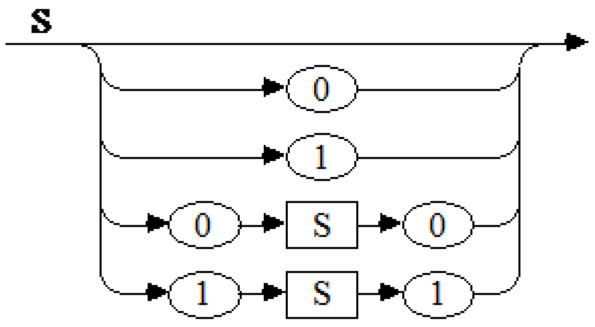


Exhibit 6.3: Syntax diagram for palindromes

An overly simple syntax for simple expressions

Why does the grammar given in previous section contain *term* and *factor*? An expression E that involves only binary operators (e.g. +, –, · and /) is either a primitive operand, abbreviated as #, or of the form 'E op E'. Consider a "simpler" grammar for simple, parenthesis-free expressions (Exhibit 6.4):

```
E = '#' | E ( '+' | '-' | '·' | '/' ) E .
```

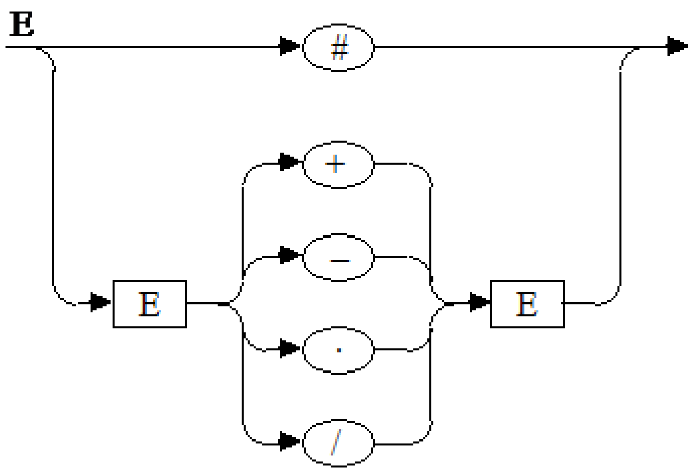


Exhibit 6.4: A syntax that generates parse trees of ambiguous structure

Now the expression # · # + # can be derived from E in two different ways (Exhibit 6.5). Such an *ambiguous* grammar is useless since we want to derive the semantic interpretation from the syntactic structure, and the tree at the left contradicts the conventional operator precedence of · over +.

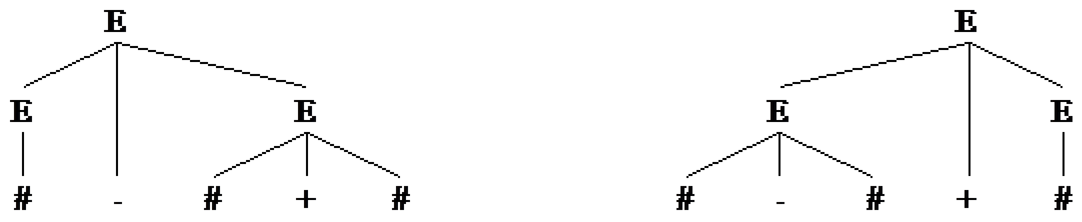


Exhibit 6.5: Two incompatible structures for the expression # · # + # .

"Everything should be explained as simply as possible, but not simpler."

(Albert Einstein)

We can salvage the idea of a grammar with a single nonterminal E by enclosing every expression of the form 'E op E' in parentheses, thus ensuring that every expression has a unique structure (Exhibit 6.6):

```
E = '#' | '(' E ( '+' | '-' | '·' | '/' ) E ')' .
```

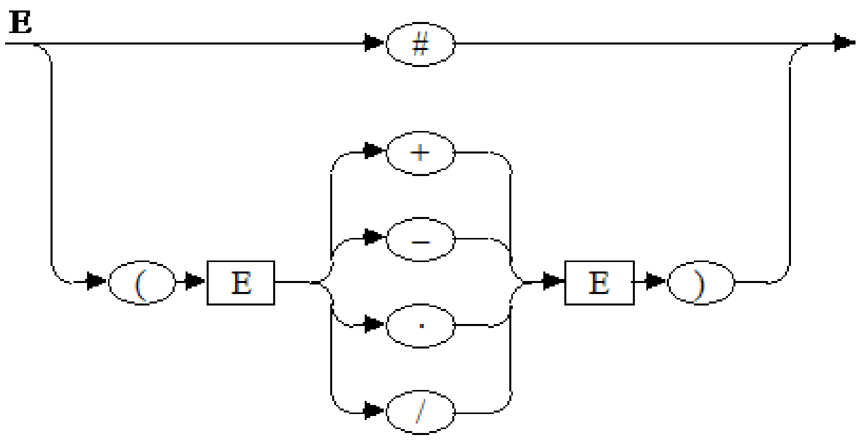


Exhibit 6.6: Parentheses serve to restore unique structure.

In doing so we change the language. The more complex grammar with three nonterminals E(xpression, T(erm), and F(actor) lets us write expressions that are only partially parenthesized and assigns to them a unique structure compatible with our priority conventions: · and / have higher priority than + and –.

Exercise: the ambiguity of the dangling "else"

The problem of the *dangling "else"* is an example of a syntax chosen to be "too simple" for the task it is supposed to handle. The syntax of several programming languages (e.g., Pascal) assigns to nested 'if-then[-else]' statements an ambiguous structure. It is left to the semantics of the language to disambiguate.

Let E, E_1, E_2, ... denote Boolean expressions, S, S_1, S_2, ... statements. Pascal syntax allows two types of if statements:

```
   if E then S
```

and

```
     if E then S else S
```

1. Draw one syntax diagram that expresses both of these syntactic possibilities.
2. Show all the possible syntactic structures of the statement

   if E_1 then if E_2 then S_1 else S_2

3. Propose a small modification to the Pascal language that avoids the syntactic ambiguity of the dangling else. Show that in your modified Pascal any arbitrarily nested structure of 'if-then' and 'if-then-else' statements must have a unique syntactic structure.

Parenthesis-free notation for arithmetic expressions

In the usual *infix* notation for arithmetic expressions a binary operator is written between its two operands. Even with operator precedence conventions, some parentheses are required to guarantee a unique syntactic structure. The selective use of parentheses complicates the syntax of infix expressions: Syntax analysis, interpretative evaluation, and code generation all become more complicated.

Parenthesis-free or Polish notation (named for the Polish logician Jan Lukasiewicz) is a simpler notation for arithmetic expressions. All operators are systematically written either before (*prefix* notation) or after (*postfix* or *suffix* notation) the operands to which they apply. We restrict our examples to the binary operators +, –, · and /. Operators with different *arities* (i.e. different numbers of arguments) are easily handled provided that the number of arguments used is uniquely determined by the operator symbol. To introduce the unary minus we simply need a different symbol than for the binary minus.

```
Infix     a+b  a+(b·c) (a+b)·c
Prefix    +ab  +a·bc    ·+abc
Postfix   ab+  abc·+   ab+c·
```

Postfix notation mirrors the sequence of operations performed during the evaluation of an expression. 'ab+' is interpreted as: load a (find first operand); load b (find the second operand); add both. The syntax of arithmetic expressions in postfix notation is determined by the following grammar:

```
S = '#' | S S ( '+' | '-' | '·' | '/' )
```

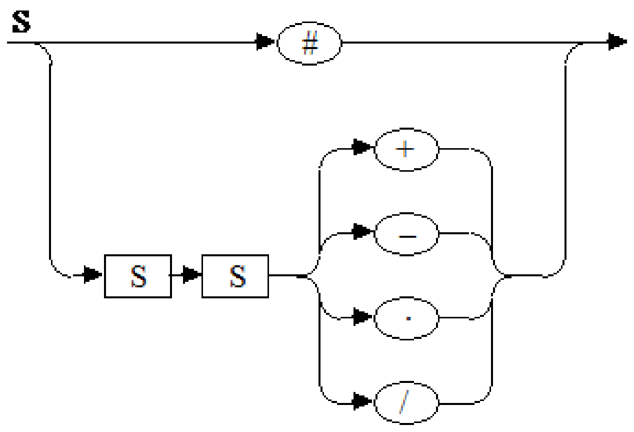


Exhibit 6.7: Suffix expressions have a unique structure even without the use of parentheses.

Exercises

1. Consider the following syntax, given in EBNF:

   S = A.

   A = B | 'IF' A 'THEN' A 'ELSE' A.

   B = C | B 'OR' C.

   C = D | C 'AND' D.

   D = 'x' | '(' A ')' | 'NOT' D.

   (a) Determine the sets of terminal and nonterminal symbols.

   (b) Give the syntax diagrams corresponding to the rules above.

   (c) Which of the following expressions is correct corresponding to the given syntax? For the correct expressions show how they can be derived from the given rules:

   x AND x

   x NOT AND x

   (x OR x) AND NOT x

   IF x AND x THEN x OR x ELSE NOT x

   x AND OR x

2. Extend the grammar of Section 6.3 to include the 'unary minus' (i.e. an arithmetic operator that turns any expression into its negative, as in –x). Do this under two different assumptions:

   (a) The unary minus is denoted by a different character than the binary minus, say ¬.

   (b) The character – is 'overloaded' (i.e. it is used to denote both unary and binary minus). For any specific occurrence of –, only the context determines which operator it designates.

3. Extended Backus-Naur form and syntax diagrams
   Define each of the four languages described below using both EBNF and syntax diagrams. Use the following conventions and notations: Uppercase letters denote nonterminal symbols. Lowercase letters and the three separators ',' '(' and ')' denote terminal symbols. "" stands for the empty or null string. Notice that the blank character does not occur in these languages, so we use it to separate distinct sentences.

L ::= a | b | … | z Letter

D ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 Digit

S ::= D { D } Sequence of digits I ::= L { L | D } Identifier

(a) Real numbers (constants) in Pascal

Examples: –3 + 3.14 10e–06 –10.0e6 but not 10e6

(b) *Nonnested lists of identifiers* (including the empty list)

Examples: () (a) (year, month, day) but not (a,(b)) and not ""

(c) *Nested lists of identifiers* (including empty lists)

Examples: in addition to the examples in part (b), we have lists such as

((),()) (a, ()) (name, (first, middle, last)) but not (a)(b) and not ""

(d) Parentheses expressions

Almost the same problem as part (c), except that we allow the null string, we omit identifiers and commas, and we allow multiple outermost pairs of parentheses.

Examples: "" () ()() ()(()) ()(()())()()

4. Use both syntax diagrams and EBNF to define the repeated if-then-else statement:

```
if B1 then S1 elsif B2 then S2 elsif … else S
```